

# Compilation Principle 编译原理

# 第12讲: 语义分析(2)

### 张献伟

# <u>xianweiz.github.io</u>

DCS290, 4/8/2021





## Review Questions (1)

- How does LALR(1) improve LR(1)? Merge similar states to reduce space.
- How to merge states?

Merge states with same core: all items are identical except lookahead

- What are the side effects of state merging? Introduce conflicts, delay error detection
- Why reduce-reduce conflict can happen in merging? Merging is the reverse of splitting, thus hurting LR(1) capability
- What are the advantages of LALR(1)? Higher parsing power than SLR(1), smaller parse table than LR(1)



### Review Questions (2)

• What are LL and LR?

LL=top-down (leftmost derivation) LR=bottom-up (reverse of rightmost derivation)

- At high level, why LR is easier or more powerful than LL?
  LR acts after seeing the entire RHS + lookahead, LR only guess with first few lookahead terminals of RHS
- Why context analysis is not performed in parsing stage? Parsing relies on CFG, which is context free.
- Give some examples of semantic analysis. Def-before-use, no redefinition, same type, scoping ...





# LALR的一些解释

- LALR(1)是LR(1)和SLR(1)的平衡
  - 文法范围: LR > LALR > SLR
  - 状态数目: LR > LALR = SLR
- 假如一个文法G是LR而非SLR
  - 依靠Follow集进行归约不够精确 --> SLR产生了冲突
    - □ 而LR通过精确的lookahead解决了冲突
  - LALR对LR进行相似状态合并
    - □ 若合并后<u>出现了冲突</u> --> 不是LALR文法
    - □ 若合并后<u>没有冲突</u> --> 是LALR文法
      - LALR可以解析文法G, 也即解决了SLR原有的冲突
      - 实际上LALR的状态数是SLR相同,但归约动作减少了(也即,对SLR解 析表而言,多个移进/归约动作的单元格中的归约被消除了)

□ 如果<u>没有相似状态</u>,则LALR=LR

- 假如一个文法G是SLR
  - 那么G一定也是LR和LALR文法
  - LR的Follow集细分是不必要的,因此LALR合并回了SLR





### Syntax Directed Translation[语法制导翻译]





## Syntax Directed Translation[语法制导翻译]

- To translate based on the program's syntactic structure[语 法结构]
  - <u>Syntactic structure</u>: structure of a program given by grammar
  - The <u>parsing process and parse trees</u> are used to direct semantic analysis and the translation of the program
    - □ i.e., CFG-driven translation [CFG驱动的翻译]
- How? Augment the grammar used in parser:
  - Attach semantic attributes[语义属性] to each grammar symbol
    - The attributes describe the symbol properties
    - An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register ...
  - For each grammar production, give **semantic rules or** actions[语义规则或动作]
    - The actions describe how to compute the attribute values associated with each symbol in a production



### Attributes[语义属性]

- Attributes can represent anything depending on the task[ 属性可以表示任意含义]
  - If computing expression: a number (value of expression)
  - If building AST: a pointer (pointer to AST for expression)
  - If generating code: a string (assembly code for expression)
  - If type checking: a type (type for expression)
- Format: X.a (X is a symbol, a is one of its attributes)
- For Project 2 Syntax Analysis
  - Semantic attributes
    - intg: string table index of integer constant value
    - *tptr*: "tree pointer" of a non-terminal symbol
  - Semantic actions
    - { ... \$\$=makeTree(ProgramOp, leftChild, rightChild); ...}



### How to Specify Syntax Directed Translation

- Syntax Directed Definitions (SDD)[语法制导定义]
  - Attributes + semantic rules[语义规则]for computing them
    - □ Attributes for grammar symbols[文法符号和语义属性关联] □ Semantic rules for productions[产生式和语义规则关联]
  - Example rules for computing the value of an expression
    - $E \rightarrow E_1 + E_2$  RULE: {E.val =  $E_1$ .val +  $E_2$ .val}
    - E -> id RULE: {E.val = id.lexval}
- Syntax Directed Translation scheme (SDT)[语法制导翻译方案]
  - Attributes + semantic actions[语义动作] for computing them
  - Example actions for computing the value of an expression

 $E \rightarrow E_1 + E_2$  {E.val =  $E_1$ .val +  $E_2$ .val} E  $\rightarrow$  id {E.val = id.lexval}



### SDD vs. SDT

- SDD[语法制导定义]: 是CFG的推广,翻译的高层次规则说明
  - A <u>CFG grammar</u> together with <u>attributes</u> and <u>semantic rules</u>
    A subset of them are also called **attribute grammars**[属性文法]
  - Semantic rules imply no order to attribute evaluation
- SDT[语法制导翻译方案]: SDD的补充,具体翻译实施方案
  - An executable specification of the SDD
    - Fragments of programs are attached to different points in the production rules
  - The order of execution is important

Grammar	SDD	SDT
D -> T L	L.inh = T.type	D -> T { L. <i>inh</i> = T. <i>type</i> } L
T -> int	T. <i>type</i> = int	T -> int { T. <i>type</i> = int }
T -> float	T. <i>type</i> = float	T -> real { T. <i>type</i> = float }
L -> L <sub>1</sub> , id	$L_1$ .inh = L.inh	L -> { L <sub>1</sub> . <i>inh</i> = L. <i>inh</i> }L <sub>1</sub> , id
<b>^</b>		



## SDD vs. SDT (cont.)

- Syntax: A ->  $\alpha$  {action<sub>1</sub>}  $\beta$  {action<sub>2</sub>}  $\gamma$  ...
- Actions are executed "at that point" in the RHS
  - *action1* executes after  $\alpha$  have been produced but before  $\beta$
  - action2 executes after  $\alpha$ , *action1*,  $\beta$  but before  $\gamma$
- Semantic rule vs. action[语义规则 vs. 语义动作]
  - SDD doesn't impose any order other than dependences
  - Location of action in RHS specifies when it should occur
    SDT specifies the execution order and time of each action



### SDD[语法制导定义]

- SDD has two types of attributes[两种属性]
  - For a non-terminal A at a parse-tree node N
- Synthesized attribute[综合属性]
  - Defined by a semantic rule associated with the production at N
    The production must have A as its head (i.e., A -> ...)
  - A synthesized attribute of node N is defined only by attribute values at N's children and N itself[子节点或自身]
- Inherited attribute[继承属性]
  - Defined by a semantic rule associated with the <u>production at</u> <u>the parent of N</u>
    - The production must have A as a symbol in its body (i.e., ... -> ...A...)
  - An inherited attributed at node N is defined only by attribute values at N's parent, N itself, and N's siblings [父节点、自身或兄弟节点]





### Synthesized Attribute[综合属性]

- Synthesized attribute for <u>non-terminal</u> A of parse-tree node N [非终结符的综合属性]
  - Only defined by N's children and N itself
    - Passed up the tree
  - P.syn\_attr = f(P.attrs, C<sub>1</sub>.attrs, C<sub>2</sub>.attrs, C<sub>3</sub>.attrs)
- <u>Terminals</u> can have synthesized attributes[终结符综合属性]
  - Lexical values supplied by the lexical analysis
  - Thus, no semantic rules in SDD for terminals









### Inherited Attribute[继承属性]

- Inherited attribute for non-terminal A of parse-tree node N[非终结符继承属性]
  - Only defined by N's parent, N's siblings and N itself
    - Passed down a parse tree
  - $-C_2$ .inh attr = f(P.attrs, C\_1.attrs, C\_2.attrs, C\_3.attrs)
- Terminals cannot have inherited attributes[终结符无继承属 性]

L.inh

- Only synthesized attributes from lexical analysis



### SDD[语法制导定义]

• Attribute dependencies in a production rule[产生式中的属性依赖]

Synthesized

- SDD has rule of the form for each grammar production b = f(A.attrs, α.attrs, β.attrs, γ.attrs)
- *b* is either an attribute in LHS (an attribute of A)

Ω

- In which case *b* is a **synthesized** attribute
- Why? From A's perspective  $\alpha$ ,  $\beta$ ,  $\gamma$  are children
- Or, *b* is an attribute in RHS (e.g., of β)
  - In which case b is an inherited attribute
  - Why? From  $\beta$ 's perspective A,  $\alpha$ ,  $\gamma$  are parent or siblings



# Example: Synthesized Attribute[综合]

#### SDD:

<b>Production Rules</b>	Semantic Rules	
(1) L -> E	print(E. <i>val</i> )	
(2) E -> E <sub>1</sub> + T	$E.val = E_1.val + T.val$	
(3) E -> T	E. <i>val</i> = T. <i>val</i>	
(4) T -> T <sub>1</sub> * F	$T.val = T_1.val \times F.val$	
(5) T -> F	T.val = F.val	
(6) F -> (E)	F.val = E.val	
(7) F -> digit	F. <i>val</i> = digit. <i>lexval</i>	

Each non-terminal has a single synthesized attribute val Terminal *digit* has a synthesized attribute lexval

#### Arithmetic expressions with + and \*

- (1) Print the numerical value of the entire expression
- (2) Compute value of summation
- (3) Value copy
- (4) Compute value of multiplication
- (5) Value copy
- (6) Value Copy





### Example: Synthesized Attribute[综合]



## Example: Inherited Attribute[继承]

#### SDD:

Production Rules Semantic Rules				
(1) D -> T L	L.inh = T.type		<i>T</i> has synthesized attribute <i>type</i> <i>L</i> has inherited attribute <i>inh</i>	
(2) T -> int	T. <i>type</i> = int			
(3) T -> float	T. <i>type</i> = float			
(4) L -> L <sub>1</sub> , id	$L_1.inh = L.inh$	Pointing	to a symbol-table object	
	addtype(id.entry)	L.inh)		
(5) L -> id	addtype(id. <i>entry</i> ,	, L. <i>inh</i> )		

#### Variable declaration of type int/float followed by a list of IDs:

- (1) Declaration: a type *T* followed by a list of *L* identifiers
- (2) Evaluate the synthesized attribute *T.type* (int)
- (3) Evaluate the synthesized attribute *T.type* (float)
- (4) Pass down type, and add type to symbol table entry for the identifier
- (5) Add type to symbol table





### Example: Inherited Attribute[继承]



float a, b, c

#### *type* depends on child *inh* depends on sibling or parent



### The Concepts

- Side effect[副作用]
  - 一般属性值计算(基于属性值或常量进行的)之外的功能
  - 例如: code generation, print results, modify symbol table ...
- Attribute grammar[属性文法]
  - 一个没有副作用的SDD
  - The rules define the value of an attribute purely in terms of the value of other attributes and constants[属性文法的规则仅仅 通过其他属性值和常量来定义一个属性值]
- Annotated parse-tree[标注分析树]
  - 每个节点都带有属性值的分析树
    - A parse tree showing the value(s) of its attribute(s)
  - a.k.a., attribute parse tree[属性分析树]



## Dependence Graph[依赖图]

- Dependence relationship[依赖关系]
  - Before evaluating an attribute at a node of a parse tree, we must evaluate all attributes it depends on
- Dependency graph[依赖图]
  - While the annotated parse tree shows the values of attributes, a dependency graph helps determine how those values can be computed[依赖图决定属性值的计算]
  - Depicts the flow of info among the attribute instances in a particular parse tree[描绘了分析树的属性信息流]
    - Directed graph where edges are dependence relationships between attributes
    - For each parse-tree node X, there's a graph node for each attr of X
    - If attr X.a depends on attr Y.b, then there's one directed edge from X.a to Y.b



# Example: Dependency Graph



float a, b, c

'entry' is dummy attribute for the addtype()





### Evaluation Order[属性值计算顺序]

- Ordering the evaluation of attributes[计算顺序]
  - Dependency graph characterizes possible orders in which we can evaluate the attributes at the various nodes of a parse-tree
- If the graph has an edge from node *M* to node *N*, then the attribute associated with *M* must be evaluated before *N*[ 用图的边来确定计算顺序]
  - Thus, the only allowable orders of evaluation are those sequences of nodes N<sub>1</sub>, N<sub>2</sub>, ..., N<sub>k</sub> such that if there is an edge of the graph from N<sub>i</sub> to N<sub>j</sub>, then i < j</li>
  - Such an ordering embeds a directed graph into a linear order, and is called a **topological sort**[拓扑排序] of the graph
    - If there's any cycle in the graph, then there are no topological sorts, i.e., no way to evaluate the SDD on this parse tree
    - □ If there are no cycles, then there is always at least one topological sort





### Example: Evaluation Order





# Evaluation Order (cont.)

- Before evaluating an attribute at a node of a parse tree, we must evaluate all attributes it depends on [依赖关系]
  - Synthesized: evaluate children first, then the node itself
    Any bottom-up order is fine
  - For SDD's with both inherited and synthesized attributes, there's no guarantee that there is even one evaluation order
- Difficult to determine whether exist any circularities[非常 难确定是否有循环依赖]
  - But, there are useful subclasses of SDD's that are sufficient to guarantee that an evaluation order exists

A.s

А

В

Such classes do not permit graphs with cycles

Production A -> B Semantic Rules A.s = B.i; B.i = A.s + 1;



### S-Attributed Definitions[s-属性定义]

- An SDD is S-attributed if every attribute is synthesized[只 具有综合属性]
- If an SDD is S-attributed (S-SDD)
  - We can evaluate its attributes in any bottom-up order of the nodes of the parse-tree[任何自底向上的顺序计算属性值]
  - Can be implemented during bottom-up parsing

Production Rules	Semantic Rules	
(1) L -> E	print(E. <i>val</i> )	
(2) E -> E <sub>1</sub> + T	$E.val = E_1.val + T.val$	
(3) E -> T	E. <i>val</i> = T. <i>val</i>	
(4) T -> T <sub>1</sub> * F	$T.val = T_1.val \times F.val$	
(5) T -> F	T. <i>val</i> = F. <i>val</i>	
(6) F -> (E)	F. <i>val</i> = E. <i>val</i>	
(7) F -> digit	F. <i>val</i> = digit. <i>lexval</i>	



